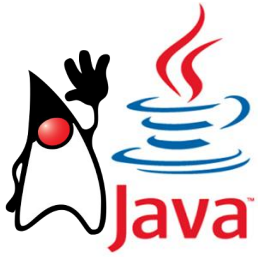
L'écosystème Java

une plateforme de développement scientifique moderne

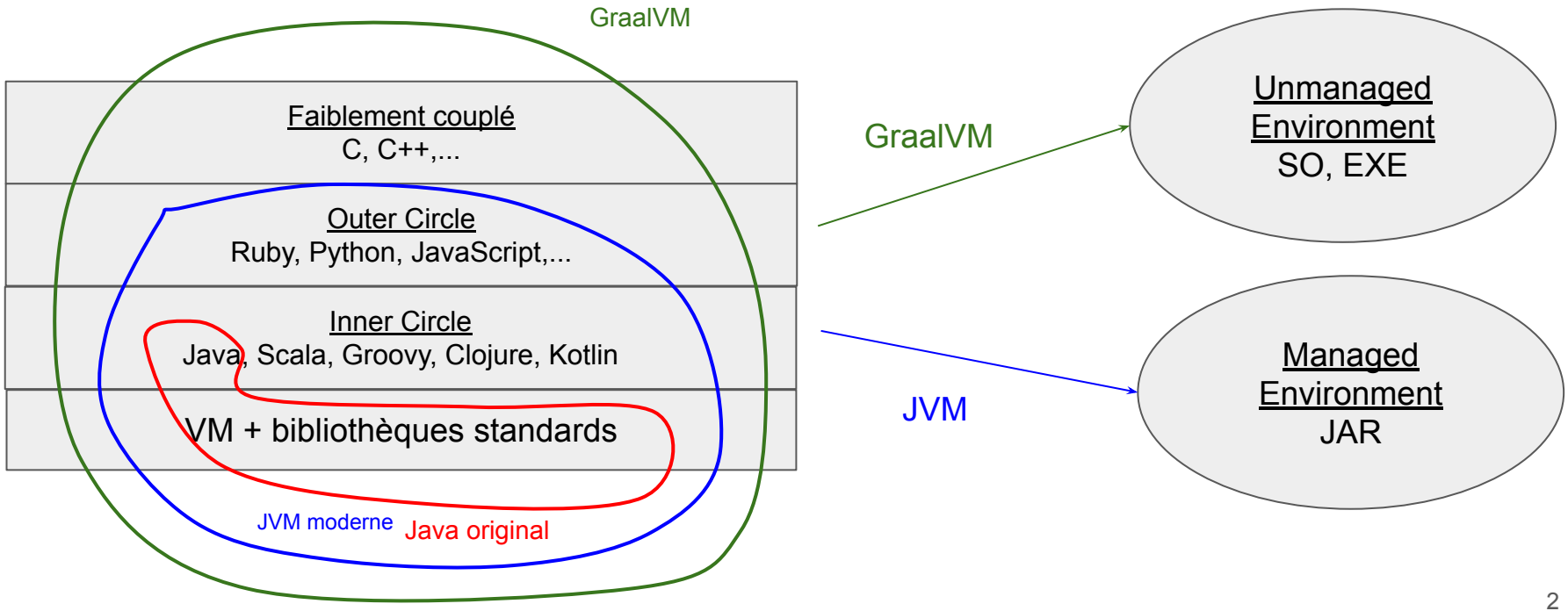


Mythe :
Java est un « simple langage »
avec de faibles performances

Réalité :
Java est un écosystème multi-langage mature
avec plusieurs modèles d'exécution et de
bonnes performances pour de nombreux cas réels

Julius Hrivnac, IJCLab
JDEV
22-25/6
Montpellier

L'écosystème Java



Java

- Environnement de programmation de haut niveau
  - Langage Java (et compilateur) + Machine Virtuelle Java (runtime) + bibliothèques standards
- Créé en 1995 chez Sun Microsystems; James Gosling a dirigé la conception originale de Java
- Principales implémentations/distributions: OpenJDK/HotSpot (implémentation de référence), Oracle JDK, Eclipse OpenJ9, GraalVM
- Évolue selon le processus formel *Java Community Process* (JSR) via les *Java Enhancement Proposals* (JEP) et les *Java Specification Requests* (JSR)
  - Toutes les fonctionnalités standards doivent avoir une implémentation de référence et un kit de compatibilité technologique (TCK)
- Deux sorties par an (mars, septembre)
  - Version actuelle : JDK 26, JDK 27 dans la version préliminaire
- Conférence annuelle Java One @ San Francisco
- Presque totalement rétrocompatible (on peut compiler/exécuter de vieux programmes en nouveau Java), sauf pour certains nouveaux mots-clés (comme `assert`)
- Environnement très dynamique et flexible (*'managed'*)
  - Réflexion, chargement de classes, gestion de la mémoire, …
- De nombreux outils de monitoring et de profilage (grâce à l'introspection)

Performance de Java

- Performance:
  - Souvent comparable à C/C++ pour les programmes dynamiques (OO) et le parallélism ou lorsque les mêmes bibliothèques de bas niveau sont utilisées (NN,...)
  - Souvent performant pour les services à longue durée car le JIT peut optimiser les "hot paths" via les données de profilage d'exécution
  - Peut être plus lent pour les outils en ligne de commande sensibles au démarrage car la VM doit démarrer et préchauffer
  - Peut nécessiter plus de mémoire que les langages non gérés car le runtime conserve des métadonnées pour le GC, le profilage, la réflexion et l'optimisation dynamique
  - La performance numérique dépend des bibliothèques et de la disposition des données; Java possède des tableaux mais pas de type de tableau multidimensionnel intégré
- Comparer les performances est très difficile:
  - Dépend de la charge de travail
  - Démarrage vs préchauffage (warmup) vs pic de performance
  - Débit (throughput) vs latence vs mémoire
  - Min vs max vs moyenne
  - L'environnement peut être réglé pour des besoins de performance spécifiques
  - Devrait comparer sur des applications **réelles**, mais comparer alors plus que le simple langage
    - Devrait inclure les fonctionnalités auxiliaires (gestion mémoire, au moins un peu de réflexion, souvent le parallélisme, ...)

Java Releases

De nombreuses nouvelles fonctionnalités

Java 25

- JEP 470: PEM Encodings of Cryptographic Objects (Preview)
- JEP 502: Stable Values (Preview)
- JEP 503: Remove the 32-bit x86 Port
- JEP 505: Structured Concurrency (Fifth Preview)
- JEP 506: Scoped Values
- JEP 507: Primitive Types in Patterns, instanceof, and switch (Third Preview)
- JEP 508: Vector API (Tenth Incubator)
- JEP 509: JFR CPU-Time Profiling (Experimental)
- JEP 510: Key Derivation Function API
- JEP 511: Module Import Declarations
- JEP 512: Compact Source Files and Instance Main Methods
- JEP 513: Flexible Constructor Bodies
- JEP 514: Ahead-of-Time Command-Line Ergonomics
- JEP 515: Ahead-of-Time Method Profiling
- JEP 518: JFR Cooperative Sampling
- JEP 519: Compact Object Headers
- JEP 520: JFR Method Timing & Tracing
- JEP 521: Generational Shenandoah

Java 26

- JEP 500: Prepare to Make Final Mean Final
- JEP 504: Remove the Applet API
- JEP 516: Ahead-of-Time Object Caching with Any GC
- JEP 517: HTTP/3 for the HTTP Client API
- JEP 522: G1 GC: Improve Throughput by Reducing Synchronization
- JEP 524: PEM Encodings of Cryptographic Objects (Second Preview)
- JEP 525: Structured Concurrency (Sixth Preview)
- JEP 526: Lazy Constants (Second Preview)
- JEP 529: Vector API (Eleventh Incubator)
- JEP 530: Primitive Types in Patterns, instanceof, and switch (Fourth Preview)

Java Releases

Java 26+

- Project **Valhalla**: *Value classes*, whose objects lack identity, but can in certain cases get an improved memory layout (with less indirection), or have their allocation optimized away entirely.
- Project **Panama**:
  - *Improved interoperability with native code*, to enable Java source code to call functions and use data types from other languages, in a way that is easier and has better performance than today.
  - Vector API, a portable and relatively low-level abstraction layer for SIMD programming. Its stabilization is dependent on Project Valhalla.
- Project **Lilliput**: *Reduce the size of Java object headers.* First down to 64 bits, and then down to 32 bits.
- Reducing startup time and warm-up time (time to peak performance) in JIT mode:
  - Project CRaC enables making snapshots of whole JVM (together with the running application) and restoring it with necessary adjustments (reopening files, sockets, etc).
  - Project Leyden, among other things, will allow partial or (in the long term) full AOT compiling, reducing overall dynamism (by adopting so called "closed-world constraints") to reduce dynamic compiling overhead.
- Project **Babylon** aims to extend the Java language's reach to alternative programming models with an enhancement to its reflective programming abilities, called code reflection (i.e., reflection over code itself). The stated main goal is to run Java code on GPUs, with SQL and other programming models as secondary targets.

Java Projets

Projet Valhalla

Value Objects & Memory Flattening

Aplatissement de la mémoire pour éliminer l'indirection des pointeurs. Efficacité cache-CPU comparable au C/C++.

Projet Panama

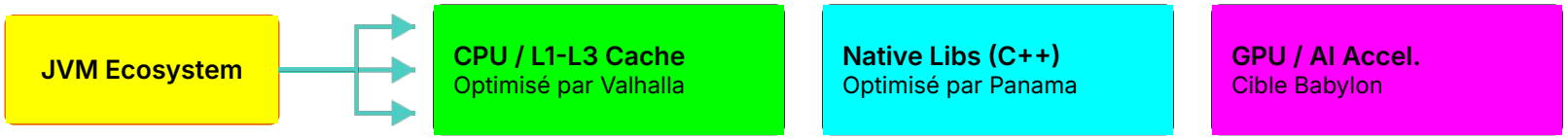
Foreign Function & Memory API

Interopérabilité native accélérée. Remplace JNI pour un accès performant au ML et graphisme natif.

Projet Babylon

GPU & Hardware Accelerators

Extention de Java aux GPU et FPGA. Idéal pour l'IA et le calcul intensif directement depuis la JVM.



Modèle d'Objet Java

- Mécanisme très sophistiqué pour créer des Objets à partir de différentes sources via une hiérarchie de `ClassLoaders` (ce que fait `'new'`)
- Permet de construire des Objets comme des Lego
  - Classes système
  - À partir de fichiers JAR
  - Depuis le réseau
    - En tant que Java Beans (Web Service)
  - Via la sérialisation, bases de données d'objets (ex: lecture de fichiers Root)
  - Utilisation d'**Aspects** (= amélioration des objets au moment de l'exécution)

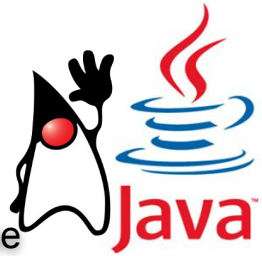
```
ClassLoader loader = new MyClassLoader(...);
Object o = loader.loadClass("MyNamespace.MyClass").newInstance();
```

- Le nom complet de la classe inclut l'espace de noms du classloader + le nom de la classe
  - Nous pouvons donc avoir différentes classes avec le même nom dans un seul programme
    - Permet la migration d'objets (= un objet change de classe)
    - Permet la modification dynamique des classes
- Base pour la réflexion, la gestion de la mémoire, ...
- Peut être complexe et non intuitif à utiliser (ex: modèle anti-héritage)
  - Parfois trop utilisé (log4j ?)
  - Le développeur d'applications en a rarement besoin
- Depuis Java 9, étendu aux Modules Java (qui peuvent explicitement importer/exporter/masquer des composants) pour éviter "dependency diamonds"
- Fondation pour un environnement multi-langages
  - Classloaders chargeant à partir de différents langages dans le même runtime

Environnement Multi-langages de la JVM

- Langages complètement interopérables avec Java (chargés dans le même runtime ou compilés en fichiers de classes standard)
  - Groovy, Scala, Kotlin, Clojure (et Java)
- Langages d'origines différentes, rendus interopérables par ré-implémentation (ou via des ponts spécifiques)
  - Go, Haskel, JavaScript, Lisp, OCaml, Pascal, PHP, Python, R, Rexx, Ruby, Scheme, Smalltalk, Tcl,...
    - Plus de 100 languages disponibles d'une manière ou d'une autre
- Les langages de bas niveau (C, C++, ...) ne peuvent être connectés que via JNI ou JNA directs
  - Car ils s'exécutent dans un **environnement non géré**
  - *Cela change avec **GraaLVM***

Langages JVM



- Langages complètement interopérables avec Java (chargés dans le même runtime ou compilés en fichiers de classes standard)
- Entièrement interopérables
- On peut librement mélanger du code de tous ces langages (même via l'héritage)
- Peuvent être utilisés de manière scriptée, interprétée ou compilé
- Les nouvelles fonctionnalités réussies de ces langages sont incorporées dans Java lui-même (ex: syntaxe fonctionnelle de Scala, style de script de Groovy)

- **Groovy** (Apache): langage de script de très haut niveau
- **Scala** (Apache): langage fonctionnel, utilisé dans Spark
- **Kotlin** (Google): pour Android
- **Clojure**: de type Lisp

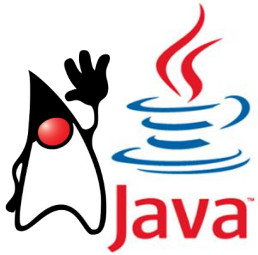
Langages JVM - Groovy, Gremlin

- **Langage de script** complètement compatible avec Java
  - Peut être considéré comme une extension de Java
- Peut mélanger librement du code Java et Groovy
  - `groovyc` peut compiler du code Java
  - Parfois des différences sémantiques (identité, type dispatch,...)
- Syntaxe extensible
  - par exemple, ***Gremlin*** pour naviguer dans les structures de graphes
- ***Grails***: framework d'application (basé sur Spring Boot)


```
g.V().has('lbl', 'OCol').
      has('classifier', 'FINK').
      group().
      by(values('cls')).
      by(out().count()).
      unfold()
```


```
#!/usr/bin/env groovy
// conversion de SQL en XML avec Groovy
// exécuté soit comme un script shell, soit compilé
// —-----------------------------------------------------
sql = Sql.newInstance("jdbc:mysql://localhost/Tuples",
                      "org.gjt.mm.mysql.Driver")
xml = new MarkupBuilder(new File("Tuples.xml"))
xml.tagSet() {
  sql.eachRow("select * from tuple where run > 2") {
    row -> xml.tag(Run:row.run, Event:row.event)
    }
  }
```

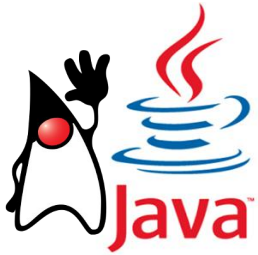
Environnement de Construction/Déploiement



- Généralement géré par ***Maven*** (fichiers de conf basés sur XML)
- Packages stockés dans des dépôts Maven
- Peut être intégré de manière transparente via ***Gradle*** (basé sur Groovy)

```
@GrabResolver(name='hrivnac', root='https://raw.githubusercontent.com/hrivnac/Maven/main/')
@GrabResolver(name='central', root='https://repo1.maven.org/maven2/')
@Grab(group='com.hrivnac', module='Lomikel', version='03.09.00')
@Grab(group='org.apache.logging.log4j', module='log4j-core', version='2.23.1')
```

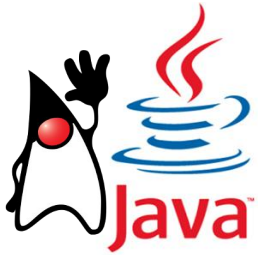
GraalVM



- Polyglot (J)DK & (J)VM
- Par Oracle
  - Gros effort
  - Également inclus dans OracleDB
  - Déjà utilisé dans l'industrie
- CE (Community Edition) : licence GPL (ou moins) - comme Java
  - Les composants ont les mêmes licences que les implémentations originales (ex. GrallVM-Python comme Python)
- EE (Enterprise Edition) : meilleures performances, sécurité, support, ...
- GraalVM JIT est inclus dans OpenJDK : `java -XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler`
  - Facile à essayer
- Suit les versions de la JVM
- Linux, MS, MacOSX
- Projet similaire dans le passé : NestedVM - a échoué en 2009

GraalVM

Fonctionnalités majeures



- **VM Universelle**
  - Les langages non-JVM sont au même niveau que les langages JVM (=> interopérabilité totale)
  - Tous les langages s'exécutent dans la même VM (les environnements multi-langages traditionnels exécutent plusieurs langages côte à côte avec des conversions fréquentes de données)
  - GraalVM est plus rapide et plus petit qu'OpenJDK (GraalVM est écrit en Java, OpenJDK est écrit en C++)
  - Interopérabilité totale entre OpenJDK et GraalVM (un programme compilé pour l'un peut s'exécuter dans l'autre)
  - Peut être intégré dans des applications externes (Oracle, MySQL,...)
- **Peut construire des exécutables et des bibliothèques natifs** (en utilisant le compilateur AOT (Ahead Of Time) au lieu de JIT)
  - Entièrement interopérable avec les applications natives
  - Empreinte réduite, démarrage plus rapide, exécution parfois plus rapide
  - Perte de certaines fonctionnalités dynamiques

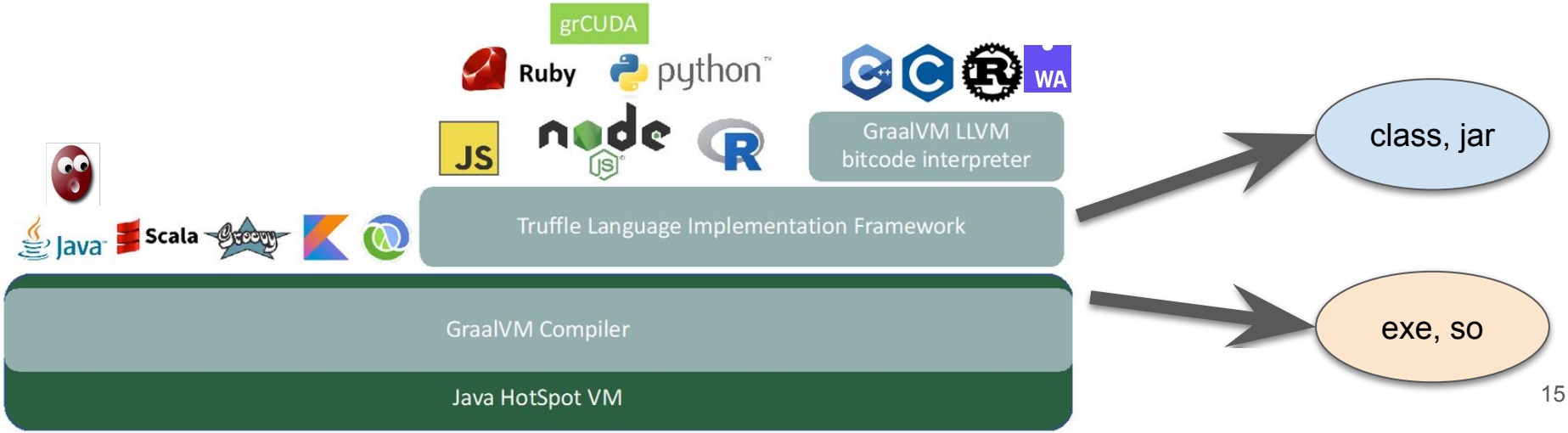
Langages supportés

- Nombre croissant de langages supportés (CUDA, WebAssembly,...)
- Nouveaux outils (débogueurs, profileurs, moniteurs,...)
- Intégration dans d'autres applications et boîtes à outils

Plusieurs langages s'exécutent dans le même espace/environnement.

X

Les pgms multi-langages traditionnels exécutent plusieurs langages côte à côte.

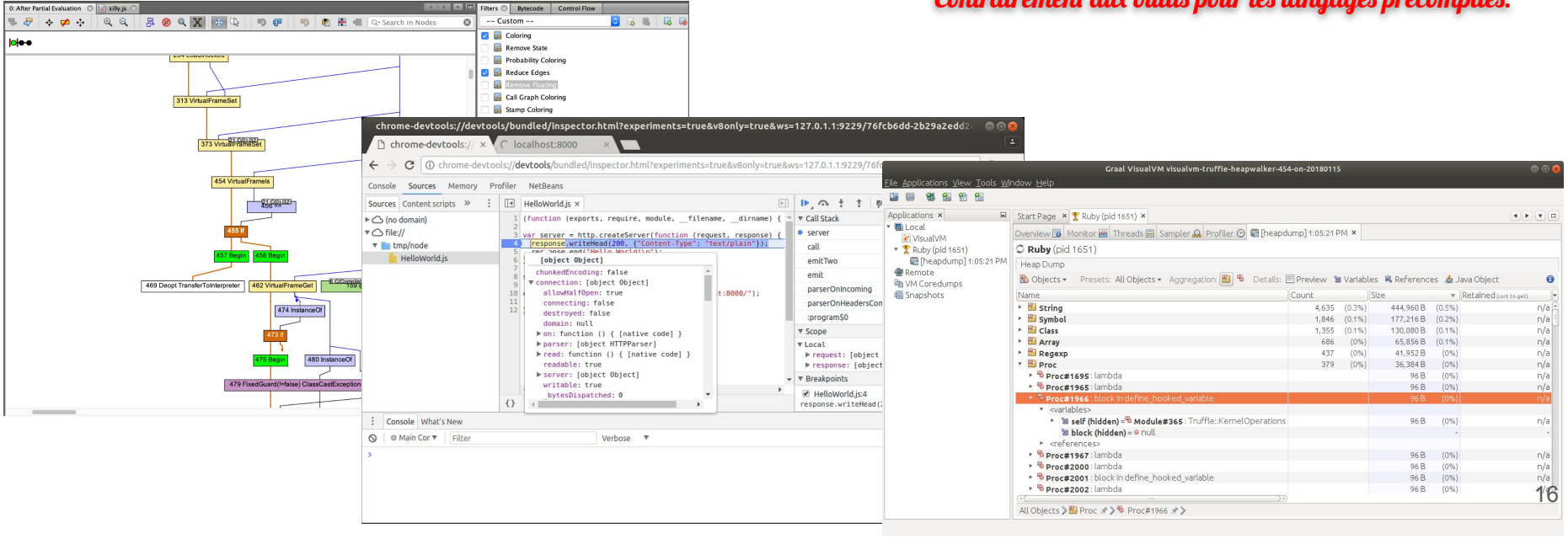


Outils

- Nombre croissant de langages supportés (CUDA, WebAssembly,...)
- Nouveaux outils (débogueurs, profileurs, moniteurs,...)
- Intégration dans d'autres applications et boîtes à outils

Les outils comprennent votre langage.

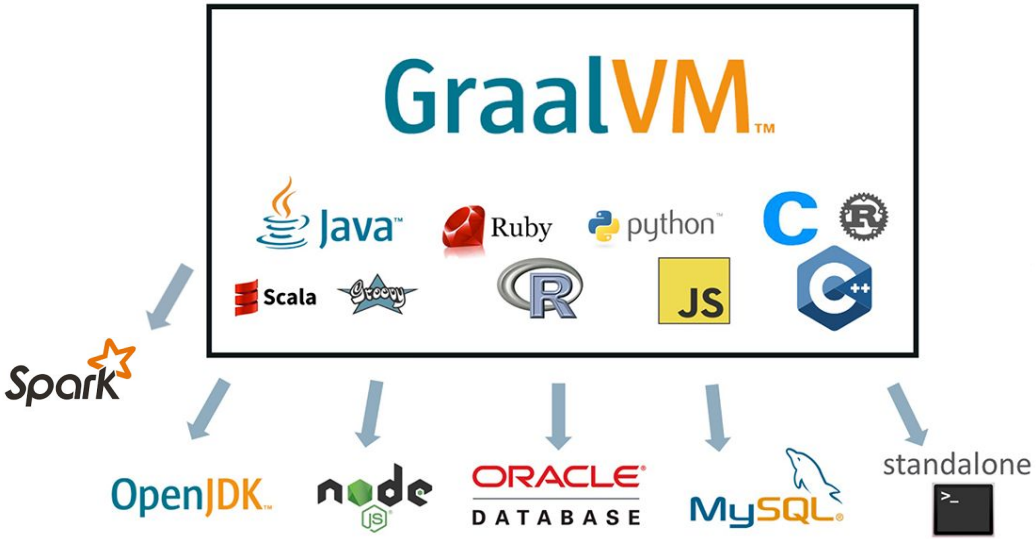
Contrairement aux outils pour les langages précompilés.



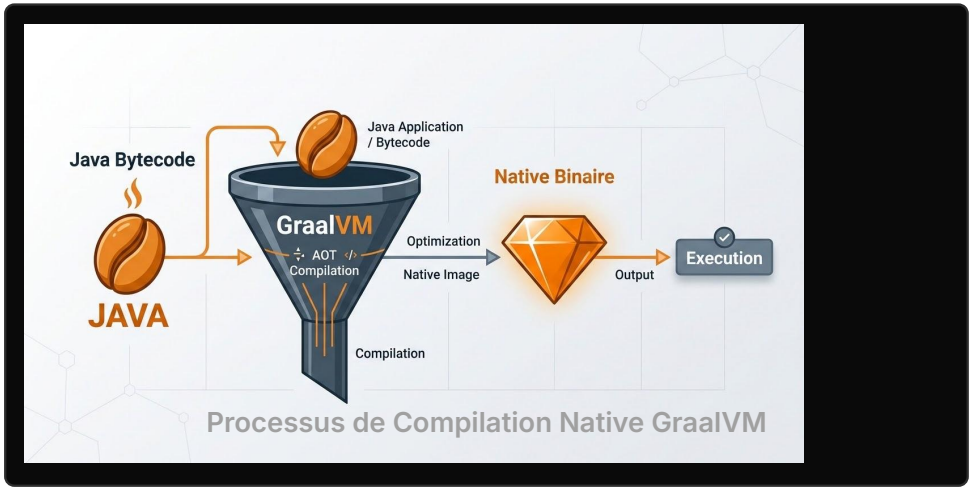
Intégration

- Nombre croissant de langages supportés (CUDA, WebAssembly,...)
- Nouveaux outils (débogueurs, profileurs, moniteurs,...)
- Intégration dans d'autres applications et boîtes à outils

Permet, par exemple, d'utiliser MySQL avec Python au lieu de SQL



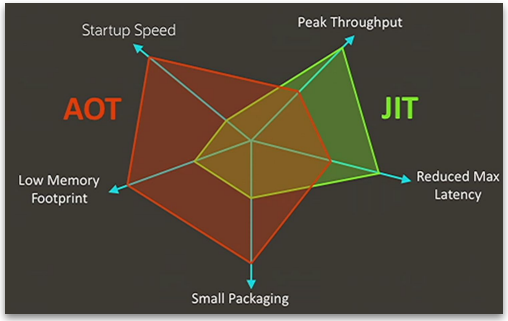
Exécutables natifs

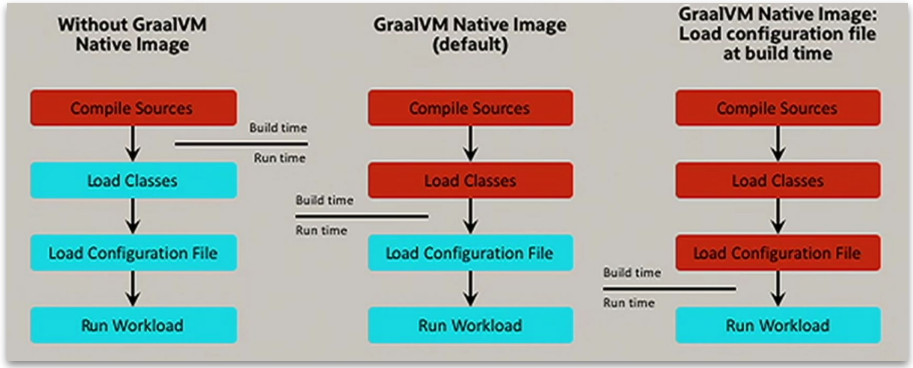


JIT vs AOT

Générer une image native GraalVM est préférable à une réécriture de Java/Python/... en C/C++/Go,..

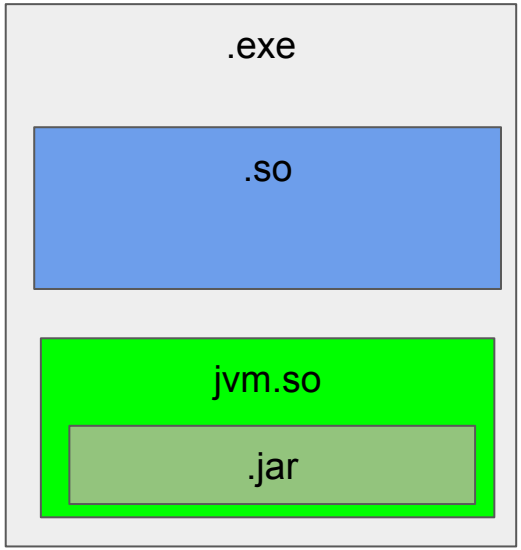
- JIT = Compilateur Just In Time: compilation en bytecode (jar), recompilation dynamique à l'exécution par la JVM (HotSpot)
- AOT = Compilateur Ahead Of Time: compilation en binaire natif (exe, so)
  - Très complexe en raison de la nature extrêmement dynamique de Java - tente de deviner ce qui se passe à l'exécution
  - Exécute l'initialisation et crée le heap initial pendant la compilation
  - *Hypothèse du monde clos (Close World Assumption)*: toutes les dépendances doivent être disponibles à la compilation (pas vrai pour le JIT), pas de chargement dynamique
  - Peut nécessiter des indices sur l'utilisation dynamique (opérations de réflexion, initialisation de classe, lambdas, annotations, chargeurs de services, ...)
    - Peut utiliser l'**Agent de traçage** pour cela
    - Peut placer cette configuration dans le jar `META-IN/native-image`
  - Peut être configuré pour ajuster l'image (mémoire vs vitesse, ...)
  - Peut nécessiter la JVM à l'exécution (*image de secours*) pour gérer certaines opérations dynamiques
- Peut compiler un jar en exe, so





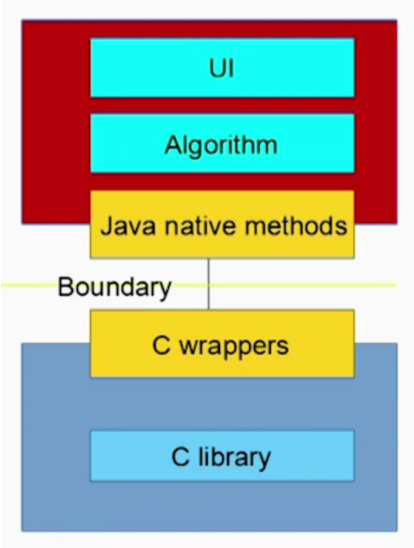
JIT vs AOT

- Certaines constructions Java ne peuvent pas être compilées en image native
  - Car elles peuvent utiliser des fonctionnalités dynamiques non disponibles dans les environnements non gérés
- La compilation en image native peut compromettre les performances
  - Structure de la mémoire OO
  - Traitement parallèle
- Nous pouvons donc décider de laisser des parties du code en Java et d'inclure la JVM dans l'image native
  - La JVM elle-même est une image native (compilée à partir du source Java)
  - Nous créons ainsi une image native complète
- L'utilisateur peut décider où placer la frontière entre
  - L'image native (mémoire réduite, démarrage plus rapide, parfois meilleures performances)
  - Java géré par une JVM intégrée (plus dynamique, souvent meilleures performances)
- La division peut être spécifiée
  - Manuellement
  - À l'aide d'outils de profilage

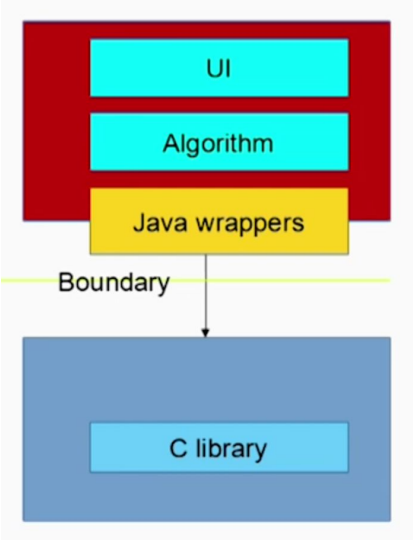


GraalVM

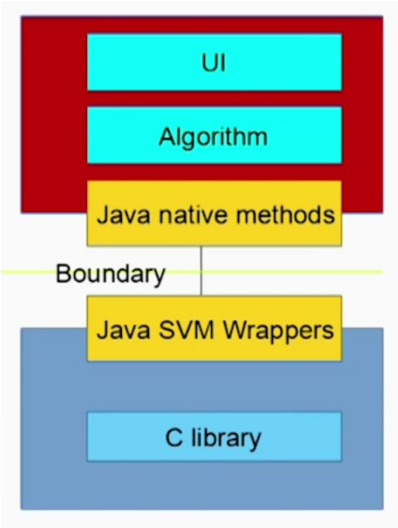
Java appelle C



JNI Traditionnel
lent, complexe



JNA Traditionnel
plus rapide, complexe



JNI via Native Image
rapide, plus simple

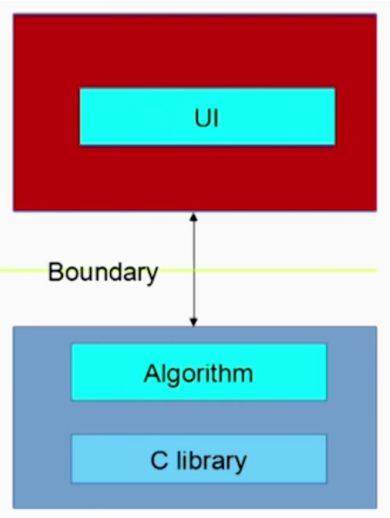


Image Native
rapide, simple

Exemple d'Image Native

Exemple de Base

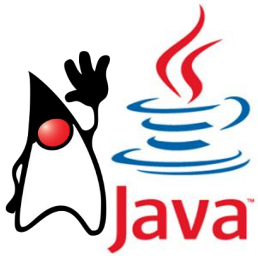
```
$ javac Hello.java
$ time java Hello
Hello !
0,10s user 0,03s system 131% cpu 0,097 total
$ native-image Hello
========================================================================================================================
GraalVM Native Image: Generating 'hello'...
========================================================================================================================
[1/7] Initializing...                                                                                    (4.1s @ 0.21GB)
 Version info: 'GraalVM 22.0.0.2 Java 11 CE'
[2/7] Performing analysis...  [*******]                                                                 (12.7s @ 0.47GB)
   2,563 (82.60%) of  3,103 classes reachable
   3,211 (60.36%) of  5,320 fields reachable
  11,648 (72.43%) of 16,082 methods reachable
      27 classes,     0 fields, and   135 methods registered for reflection
      57 classes,    58 fields, and    51 methods registered for JNI access
[3/7] Building universe...                                                                               (0.8s @ 0.62GB)
[4/7] Parsing methods...      [*]                                                                        (0.8s @ 0.84GB)
[5/7] Inlining methods...     [****]                                                                     (1.2s @ 0.75GB)
[6/7] Compiling methods...    [***]                                                                      (9.3s @ 1.19GB)
[7/7] Creating image...                                                                                  (1.1s @ 1.45GB)
   3.69MB (35.06%) for code area:    6,949 compilation units
   5.86MB (55.66%) for image heap:   1,543 classes and 80,509 objects
 999.26KB ( 9.28%) for other data
  10.52MB in total
------------------------------------------------------------------------------------------------------------------------
Top 10 packages in code area:                               Top 10 object types in image heap:
 606.25KB java.util                                           1.64MB byte[] for general heap data
 282.31KB java.lang                                         715.44KB java.lang.String
 222.52KB java.util.regex                                   548.99KB java.lang.Class
 219.55KB java.text                                         451.55KB byte[] for java.lang.String
 193.17KB com.oracle.svm.jni                                363.75KB java.util.HashMap$Node
 149.73KB java.util.concurrent                              192.00KB java.util.HashMap$Node[]
 117.92KB java.math                                         139.81KB java.lang.String[]
 103.60KB com.oracle.svm.core.reflect                       139.04KB char[]
  97.83KB sun.text.normalizer                               130.59KB java.util.concurrent.ConcurrentHashMap$Node
  88.78KB com.oracle.svm.core.genscavenge                   103.92KB sun.util.locale.LocaleObjectCache$CacheEntry
     ... 111 additional packages                                ... 723 additional object types
                                        (use GraalVM Dashboard to see all)
------------------------------------------------------------------------------------------------------------------------
                     1.6s (5.1% of total time) in 17 GCs | Peak RSS: 2.54GB | CPU load: 3.33
------------------------------------------------------------------------------------------------------------------------
Produced artifacts:
 hello (executable)
 hello.build_artifacts.txt
========================================================================================================================
Finished generating 'hello' in 31.1s.
$ time hello
Hello !
0,00s user 0,00s system 89% cpu 0,002 total
```

Exemple Réel

```
${graalvm_dir}/bin/native-image \
--delay-class-initialization-to-runtime=\
io.grpc.netty.shaded.io.netty.handler.ssl.OpenSsl \
--initialize-at-build-time=\
org.apache.log4j.Level,\
org.apache.log4j.Layout,\
org.apache.log4j.PatternLayout,\
org.apache.log4j.Logger,\
org.apache.log4j.helpers.LogLoorg.apache.log4j.Level,\
org.apache.log4j.Priority,\
org.apache.log4j.LogManager,\
org.apache.log4j.helpers.Loader,\
org.apache.log4j.helpers.LogLog,\
org.apache.log4j.Category,\
org.apache.log4j.spi.RootLogger,\
org.apache.log4j.spi.LoggingEvent,\
org.slf4j.LoggerFactory,\
org.slf4j.impl.Log4jLoggerAdapter,\
org.slf4j.impl.StaticLoggerBinder,\
java.beans.Introspector,\
com.sun.beans.Introspector,\
com.sun.beans.introspect.ClassInfo \
--report-unsupported-elements-at-runtime \
-H:Name=GroovyEL.exe \
-H:Path=../bin \
-jar ../lib/GroovyEL.exe.jar
```

GraalVM

Exemples Polyglottes (1)



- Les objets ne sont jamais copiés
- Conversion (au format physique client) au moment le plus tardif possible
- Tous les outils sont disponibles pour tous les langages
- **Plusieurs façons d'appeler un langage étranger :**
  - **Charger comme script et exécuter**
  - **Compiler en tant que classe et utiliser**
  - **Générer une Image Native et l'appeler**

```
// Java calling C
Context context = Context.create();
File file = new File("polyglot"); // c-pgm compiled with GraalVM
Source source = Source.newBuilder("llvm", file).build();
Value cpart = polyglot.eval(source);
cpart.execute();
```

```
// Java calling Python
Value clazz = context.eval(Source.newBuilder("python", new File("mycode.py")).build());
Value instance = clazz.newInstance(1234);
System.out.println(instance.invokeMember("pyMethod", new int[]{1, 2, 3}));
```

```
// C calling JS
poly_create_context(thd, &ctx);
poly_context_eval(thd, ctx, "js", "foo", "function() {return 42;}", &func);
poly_value_execute(thd, func, NULL, 0, &answer);
poly_value_fits_in_int32(thd, answer, &fits);
poly_value_as_int32(thd, answer, &result);
return result;
```

```
// Java calling JS
Context context = Context.create();
Value v = context.eval("js", "function() {return 42;}");
Value answer = v.execute();
return answer.asInt();
```

Exemples Polyglottes (2)

- L'interaction avec les langages LLVM nécessite plus de code standard ("boilerplate")
- Il est plus simple de compiler du code JVM en Image Native que d'interfacer la JVM avec LLVM
- L'appel de Java par C++ est plus simple que l'appel de C++ par Java

```
// JS calls CUDA
const DeviceArray = Polyglot.eval('grcuda', string='DeviceArray')
const in_arr = DeviceArray('float', 1000)
const out_arr = DeviceArray('float', 1000)
// set arrays ...
const code = '__global__ void inc_kernel(...) ...'
const buildkernel = Polyglot.eval('grcuda', string='buildkernel')
const incKernel = buildkernel(code, 'inc_kernel', 'pointer, pointer, uint64')
incKernel(160, 256)(out_arr, in_arr, N)
```

```
// C++ calls Java

// C++
int main() {
  graal_isolate_t *isolate = NULL;
  graal_isolatethread_t *thread = NULL;
  graal_create_isolate(NULL, &isolate, &thread);
  printf("Result> %d\n",ceilingPowerOfTwo(thread, 14));
  }

// Java
public class MyMath {
  @CEntryPoint (name = "ceilingPowerOfTwo")
  public static int ceilingPowerOfTwo(IsolateThread thread, int x) {
    return IntMath.ceilingPowerOfTwo(x);
    }
  }
```

```
// JS calls C++

// JS
loadSource("llvm", "cpppart");
Value getSumOfArrayFn = polyglotCtx.getBindings("llvm").getMember("getSumOfArray");
int sum = getSumOfArrayFn.execute(sqrNumbers, sqrNumbers.length).asInt();

// C++
extern "C" getSumOfArray(int array[], int size) {
  int i, sum = 0;
  for (i = 0; i < size; i++) {
    sum += array[i];
    }
  return sum;
  }
```

Où l'utiliser

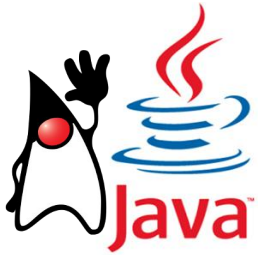
- Bonne nouvelle: **Le simple fait d'utiliser GraalVM JIT (inclus dans OpenVM) le rend plus rapide (il peut optimiser davantage que le compilateur Java standard)**
- Pour les langages JVM :
  - La compilation avec le compilateur GraalVM peut aider
  - La création d'une Image Native peut améliorer les performances
  - Permet une meilleure intégration avec d'autres langages
  - Pour Scala :
    - GraalVM JIT est capable d'optimiser Scala bien plus qu'OpenVM JIT (facteur > 2)
- Pour Python :
  - Interopérabilité complète avec les langages JVM
  - Vitesse, surtout lors de la compilation en Image Native
  - Meilleure interopérabilité avec C/C++ lors de la compilation en Image Native
- Pour C/C++ :
  - Peut remplacer du code C/C++ par du code dans de meilleurs langages ou intégrer des composants existants écrits dans de meilleurs langages
    - En les compilant en Image Native ou en les connectant à l'environnement multi-langage Truffle
  - Intégration dans des frameworks écrits dans d'autres langages
  - Possibilité de s'exécuter dans un *Environnement Géré* (donc débogage facile)
  - Parfois un gain de performance juste en reconstruisant avec GraalVM (sans modification)

```
graalpy -m standalone native
        –module my_script.py
        –output my_binary
        –venv my_env
```

On peut réécrire une seule partie du système dans un autre langage (plus adapté), et compiler en exécutable natif.

GraalVM

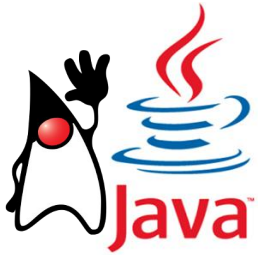
Limitations Intrinsèques



- Peut être complexe à configurer
  - Dans de nombreux cas, la génération d'images natives doit être configurée/ajustée
  - On peut/doit configurer/ajuster pour la performance
- Certaines applications (Java) peuvent avoir besoin de la JVM même compilées en exécutable natif
  - Lorsqu'elles (mal)utilisent la réflexion et construisent des classes à l'exécution
    - Par exemple log4J
  - Mais après tout, on peut considérer la JVM comme une autre bibliothèque native (ce qu'elle est)
- On peut gagner en vitesse pour les petites applications, moins souvent pour les grandes complexes
  - Pas surprenant, Java est souvent rapide pour les applications réelles
- En compilant en exécutable natif, on perd en flexibilité et en portabilité
- Les langages Truffle (Python, Ruby, JS,...) ne sont pas au même niveau d'interopérabilité que les langages JVM directs
- La coexistence des langages LLVM (C, C++, Rust) avec les langages JVM n'est pas aussi simple qu'entre deux langages JVM
  - Modèles de mémoire et d'objets différents
  - Les valeurs, objets, noms doivent être convertis
  - Une communication lourde à travers la frontière LLVM-JVM peut ralentir l'exécution
  - Dans ce cas, il peut être plus utile de compiler les langages JVM en image native
  - Mais c'est probablement le maximum possible pour intégrer les langages JVM et C

GraalVM

Complications Externes

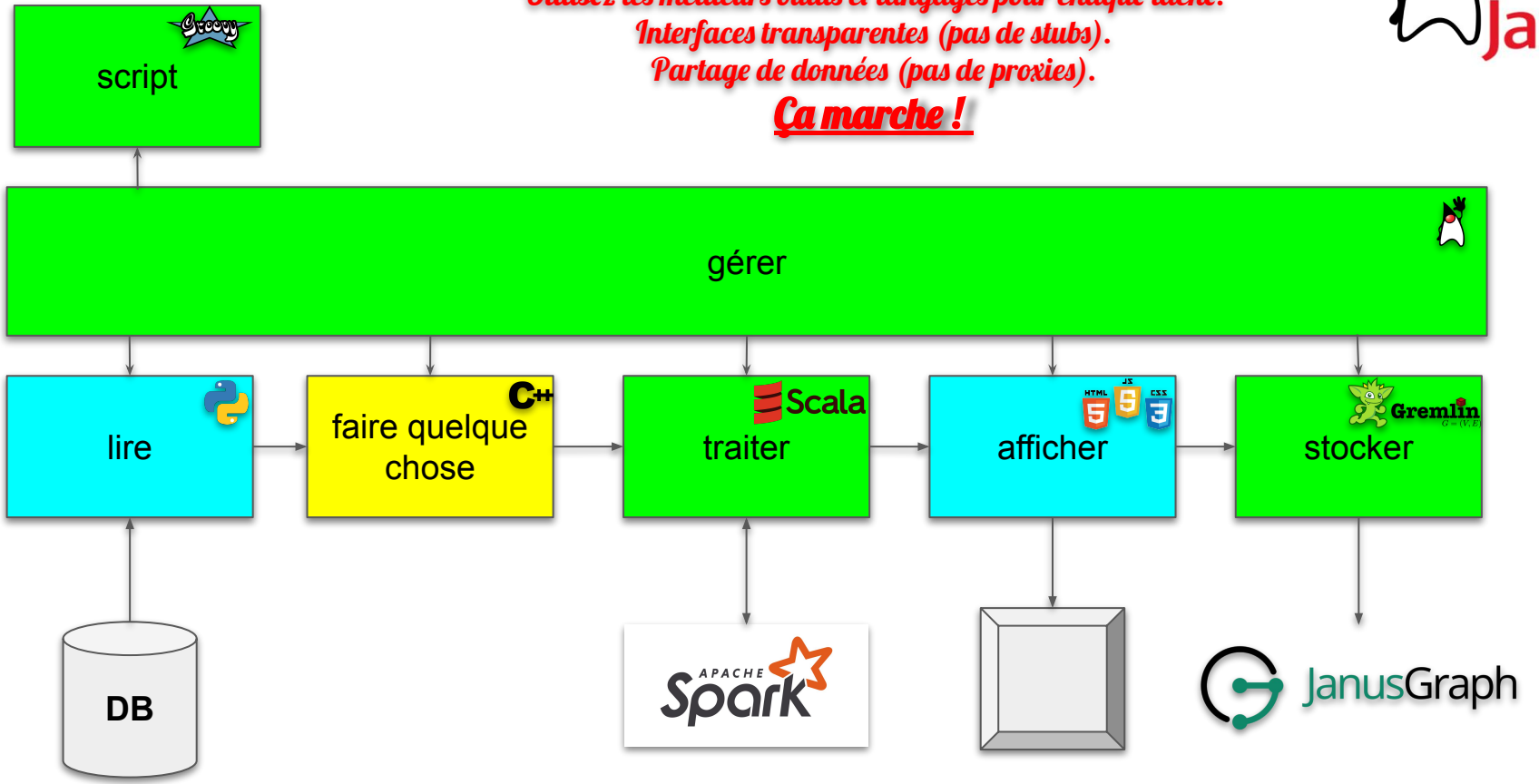


- Systèmes de construction spécifiques au langage
  - Fichiers *make* très élaborés
- Systèmes de déploiement spécifiques au langage
  - Installation silencieuse des dépendances
    - Pip, conda, node, …
- Passerelles spécifiques entre les langages
  - Souvent, l'implémentation interne utilise d'autres langages
    - Les paquets Python contiennent souvent du code C, …
- Versions de langage
  - Il est impossible de supporter toutes les versions et tous les dialectes
    - Python 2 vs 3, …
- Environnements complexes spécifiques au projet

Longue liste de projets qui ont déjà été portés/migrés/interfacés.
Les plus populaires et les moins propriétaires.

Application Multilingue Idéale

Utilisez les meilleurs outils et langages pour chaque tâche.
Interfaces transparentes (pas de stubs).
Partage de données (pas de proxies).
Ça marche !

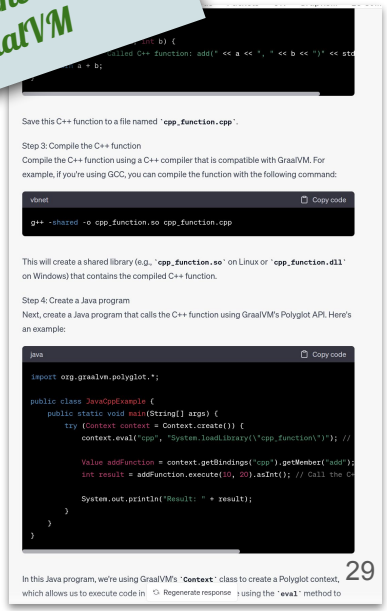


L'Avenir de la Programmation

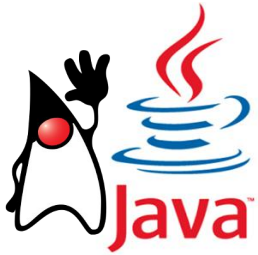
- Les frameworks seront constitués de divers composants...
  - Boîtes noires tierces
  - Écrit par l'IA
  - Boîtes héritées (legacy)
- Parfois, nous ne saurons même pas quel est le langage d'implémentation
  - Cela fonctionne déjà dans la JVM classique
- Les langages seront utilisés pour leurs points forts (Scala pour le parallélisme, JavaScript pour le Graphisme,...)
- Transparence (plug-in) ...
- Il est important de réellement séparer les données des algorithmes et de la logique (enfin)

Possibilité de réécrire une seule partie du système dans un autre langage (plus adapté), Et de compiler en un exécutable natif.

Programme généré par ChatGPT utilisant Java & C++ connectés par GraalVM

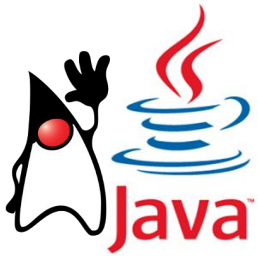


Résumé



- L'écosystème Java offre un cadre cohérent avec
  - Scripting via Groovy (ou Java)
  - Parallélisme via Scala (ou Java)
  - Environnement de construction et de déploiement via Gradle
  - GUI via JavaScript ou JavaFX
  - Performance
    - OO et parallélisme directement avec Java
    - Interfaces de haut niveau vers des packages C/C++ de bas niveau (comme Keras, Tensorflow, PyTorch,...)
    - Compilation AOT en image native lorsque c'est utile
  - Large choix de packages disponibles
    - Java lui-même (souvent Apache)
    - Tout le code Python disponible directement (souvent plus rapide qu'en utilisant directement Python)
  - Support serveur étendu
- *Beaucoup plus simple que C++/Python/Go/Julia/Rust !*

Java in HEP/Astro Examples



- Atlantis - ATLAS Event Display
- AMI - ATLAS Metadata Interface
- ATLAS Event Index
- Fink Broker - for ZTF & LSST Alerts